NPS: A Compiler-aware Framework of Unified Network Pruning for Beyond Real-Time Mobile Acceleration

Zhengang Li[1], Geng Yuan[1], Wei Niu[2], Yanyu Li[1], Pu Zhao[1], Yuxuan Cai[1], Xuan Shen[1], Zheng Zhan[1], Zhenglun Kong[1], Qing Jin[1], Bin Ren[2], Yanzhi Wang[1], Xue Lin[1]
[1]Northeastern University,
[2]College of William and Mary,
{li.zhen, yuan.geng, yanz.wang, xue.lin}@northeastern.edu

***Abstract*—With the increasing demand to efficiently deploy DNNs on mobile edge devices, it becomes much more important to reduce unnecessary computation and increase the execution speed. Prior methods towards this goal, including model compression and network architecture search (NAS), are largely performed independently and do not fully consider compiler-level optimizations which is a must-do for mobile acceleration. In this work, we first propose (i) a general category of fine-grained structured pruning applicable to various DNN layers, and (ii) a comprehensive, compiler automatic code generation framework supporting different DNNs and different pruning schemes, which bridge the gap of model compression and NAS. We further propose NPS, a compiler-aware unified network pruning and architecture search. To deal with large search space, we propose a meta-modeling procedure based on reinforcement learning with fast evaluation and Bayesian optimization, ensuring the total number of training epochs comparable with representative NAS frameworks. Our framework achieves 6.7ms, 5.9ms, and 3.9ms ImageNet inference times with 77%, 75% (MobileNet-V3 level), and 71% (MobileNet-V2 level) Top-1 accuracy respectively on an off-the-shelf mobile phone, consistently outperforming prior work.**

I. Introduction

The growing popularity of mobile AI applications and the demand for real-time Deep Neural Network (DNN) executions raise significant challenges for DNN accelerations. However, the ever-growing size of DNN models causes intensive computation and memory cost, which impedes the deployment on resource limited mobile devices.

DNN **weight pruning** [15], [27], [43] has been proved as an effective model compression technique that can remove redundant weights of the DNN models, thereby reducing storage and computation costs simultaneously. Existing work mainly focus on *unstructured pruning* scheme [14], [24] where arbitrary weight can be removed, and (coarse-grained) *structured pruning* scheme [27], [49] to eliminate whole filters/channels. The former results in high accuracy but limited hardware parallelism (and acceleration), while the latter is the opposite. Another active research area is the **Neural Architecture Search** (NAS) [50], which designs more efficient DNN architectures using automatic searching algorithms. EfficientNet [41] and MobileNetV3 [18] are representative lightweight networks obtained by using NAS approaches. Recently, hardware-aware NAS [7], [40], [44] has been investigated targeting acceleration on actual hardware platforms.

Different from the prior work on coarse-grained pruning and NAS that find a smaller, yet regular, DNN structure, recent work [10], [25], [31] propose to prune the weights in a more fine-grained manner, e.g., assigning potentially different patterns to kernels. Higher accuracy can be achieved as a result of the intra-kernel flexibility, while high hardware parallelism (and mobile inference acceleration) can be achieved with the assist of compiler-level code generation techniques [31]. This work reveals a new dimension of optimization: *With the aid of advanced compiler optimizations*, it is possible to achieve high accuracy and high acceleration simultaneously by injecting a proper degree of fine granularity in weight pruning. Despite the promising results, pattern-based pruning [25], [31] is only applied to 3×3 convolutional (CONV) layers, which limits the applicability.

As the **first contribution**, we propose a general category of fine-grained structured pruning schemes that can be applied to various DNN layers, i.e., *block-punched pruning* for CONV layers with different kernel sizes, and *block-based pruning* for FC layers. We develop a comprehensive, compiler-based automatic code generation framework *supporting the proposed pruning schemes in a unified manner*, *supporting other types of pruning schemes*, and *different schemes for different layers*. We show (i) the advantage of the proposed fine-grained structured pruning in both accuracy and mobile acceleration, and (ii) the superior end-to-end acceleration performance of our compiler framework on both dense (before pruning) and sparse DNN models.

While our compiler optimizations provide notable mobile acceleration and support of various sparsity schemes, it introduces *a much larger model optimization space*: Different pruning schemes result in different acceleration performances on different types of layers under compiler optimizations. Thus, it is desirable to perform a *compiler aware*, joint pruning scheme and pruning rate search for each individual layer. The *objective* is to maximize accuracy satisfying a DNN latency constraint on the target mobile device. The DNN latency will

be actually measured on the target mobile device, thanks to the fast auto-tuning capability of our compiler for efficient inference on different mobile devices.

We develop the compiler-aware NPS framework to fulfill the above goal. It consists of three phases: (1) *replacement of mobile-unfriendly operations*, (2) *the core search process*, and (3) *pruning algorithm search*. The overall latency constraint is satisfied through the synergic efforts of (i) incorporating the overall DNN latency constraint into the automatic search in Phase 2, and (ii) the effective search of pruning algorithm and performing weight training/pruning accordingly. As Phase 2 exhibits a larger search space than prior NAS work, to perform efficient search, we propose a meta-modeling procedure based on reinforcement learning (RL) with fast evaluation and Bayesian optimization. This will ensure the total number of training epochs comparable with representative NAS frameworks.

Our key contributions include:

- We propose a general category of fine-grained structured pruning applicable to various DNN layers, and a comprehensive, compiler code generation framework supporting different pruning schemes.
- We bridge the gap between model compression and NAS. We develop a compiler-aware framework of network pruning search, maximizing accuracy while satisfying inference latency constraint.
- We design a systematic search acceleration strategy, integrating pre-trained starting points, fast accuracy and latency evaluations, and Bayesian optimization.
- Our NPS framework achieves by far the best mobile acceleration: 6.7ms, 5.9ms, and 3.9ms ImageNet inference times with 77.0%, 75%, and 71% Top-1 accuracy, respectively, on an off-the-shelf mobile phone.

II. Related Works

A. Network Pruning

Existing weight pruning research can be categorized according to pruning schemes and pruning algorithms.

Pruning Scheme: Previous weight pruning work can be categorized into multiple major groups according to the pruning scheme: *unstructured pruning* [14], *coarse-grained structured pruning* [17], [24], [43], and *pattern-based pruning* [25], [31].

Unstructured pruning (Fig. 1 (a) and (b)) removes weights at arbitrary position. Though it can significantly decrease the number of weights in DNN model as a fine-grained pruning scheme, the resulted sparse and irregular weight matrix with indices damages the parallel implementations and results in limited acceleration on hardware platforms.

To overcome the limitation in unstructured, irregular weight pruning, many work [17], [24], [43] studied the coarse-grained structured pruning at the level of filters and channels as shown in Fig. 1 (c) and (d). With the elimination of filters or channels, the pruned model still maintains the network structure with high regularity which can be parallelized on hardware. The downside of coarse-grained structured pruning is the obvious accuracy degradation by removing the whole filters/channels, which limits model compression rate.

Fig. 1 (e) shows the pattern-based pruning [25], [26], [31] as a representative fine-grained structured pruning scheme. It assigns a pattern (from a predefined library) to each CONV kernel, maintaining a fixed number of weights in each kernel. As shown in the figure, each kernel reserves 4 non-zero weights (on a pattern) out of the original 3×3 kernels. Besides being assigned a pattern, a kernel can be completely removed to achieve higher compression rate. Pattern-based pruning can simultaneously achieve high accuracy (thanks to the structural flexibility) and high inference acceleration with the aid of compiler-based executable code generation. Note that **compiler support** [31] is necessary for pattern-based pruning to deliver its promise on mobile acceleration.

A limitation is that pattern-based pruning is limited to 3×3 CONV layers in current work: 5×5 or larger kernel size results in a large number of pattern types, which incurs notable computation overheads in compiler-generated executable codes. 1×1 CONV layers and FC layers leave no space of designing different patterns for a kernel.

Pruning Algorithm: Two main categories exist: *heuristic pruning algorithm* [13], [47] and *regularization-based pruning algorithm* [11], [17], [24], [43], [48]. Heuristic pruning was firstly performed in an iterative, magnitude-based manner on unstructured pruning [13], and gets improved in later work [11]. Heuristic pruning has also been incorporated into coarse-grained structured pruning [47].

Regularization-based algorithm uses mathematics-oriented method to deal with the pruning problem. Early work [17], [43] incorporates ℓ_1 or ℓ_2 regularization in loss function to solve filter/channel pruning problems. In [21], [48], an advanced optimization solution framework ADMM (Alternating Direction Methods of Multipliers) is utilized to achieve dynamic regularization penalty which significantly reduces accuracy loss. In [16], Geometric Median is proposed to conduct filter pruning.

B. Neural Architecture Search (NAS)

In general, NAS can be classified into the following categories by its searching strategy. Reinforcement Learning (RL) methods [6], [32], [50] employ Recurrent Neural Network (RNN) as predictor, with parameters updated by the accuracy of child network validated over a proxy dataset. Evolution methods [34], [45] develop a pipeline of parent initialization, population updating, generation and elimination of offsprings.

One-shot NAS [5], [12] trains a large one-shot model containing all operations and shares the weight parameters to all candidate models. Gradient-based methods [7], [23] propose a differentiable algorithm distinct from prior discrete search, reducing searching cost while still getting comparable results. Bayesian optimization [4], [35] uses optimal transport program to compute the distance of network architectures.

Some recent work realize the importance of hardware co-design and incorporate the inference latency into NAS, which

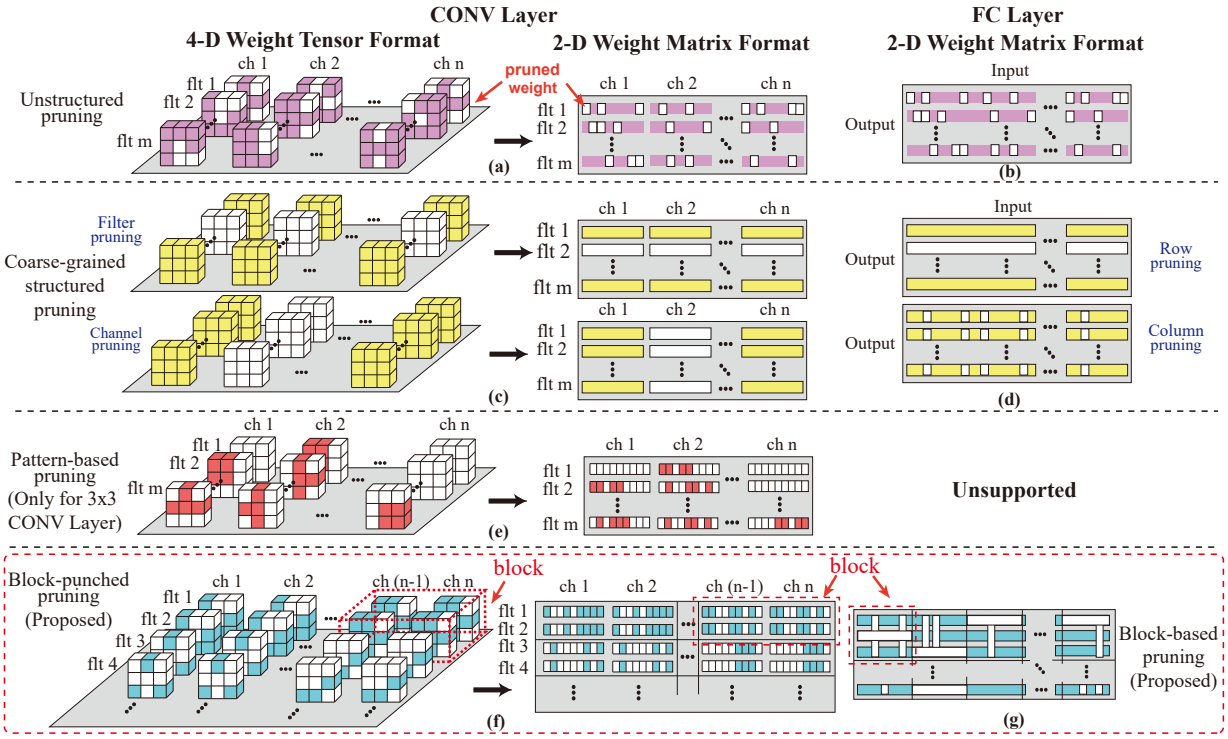

Fig. 1. Different weight pruning schemes for CONV and FC layers using 4D tensor and 2D matrix representation.

is more accurate than the intuitive volume estimation like Multiply–Accumulate operations (MACs) [7], [40], [44]. MnasNet [40] utilizes latency on mobile device as the reward to perform RL search, where gradient-based NAS work FBNet [44] and ProxylessNAS [7] add a latency term to the loss function. However, none of these hardware-targeting work fully exploit the potential of compiler optimizations or satisfy an overall latency requirement, not to mention accounting for compiler-supported sparse models. This motivates us to investigate another dimension of model optimization, that is, compiler-aware, latency-constrained, architecture and pruning co-search.

C. Compiler-assisted DNN Frameworks on Mobile

Recently, mobile-based, compiler-assisted DNN execution frameworks [20], [46] have drawn broad attention from both industry and academia. TensorFlow-Lite (TFLite) [1], Alibaba Mobile Neural Network (MNN) [2], and TVM [8] are representative state-of-the-art DNN inference frameworks. Various optimization techniques, such as varied computation graph optimizations and half-float support, have been employed to accelerate the DNN inference on mobile devices (mobile CPU and GPU).

Recent work PatDNN [31] and PCONV [25] employ a set of compiler-based optimizations to support specific pattern-based sparse DNN models to accelerate the end-to-end inference on mobile devices. However, the lack of support for different types of layers (e.g., 1×1 CONV, 5×5 CONV, and FC) limits the versatility of such framework.

III. Proposed Fine-Grained Structured Pruning

Pattern-based pruning scheme [25], [26], [31], as mentioned in Section II-A, reveals a new optimization dimension of fine-grained structured pruning that can achieve high accuracy and high inference acceleration simultaneously with the assist of compiler optimizations. As pattern-based pruning is only applicable to 3×3 CONV layers, we propose a general category of fine-grained structured pruning scheme that can be applied to various DNN layers: block-based pruning for FC layers and block-punched pruning for CONV layers with different kernel sizes.

Block-based Pruning: Fig. 1 (g) shows the block-based pruning scheme in 2D weight matrix format for FC layers. The entire weight matrix is divided into a number of equal-sized blocks, then the entire column(s) and/or row(s) of weights are pruned within each block. Compared to the coarse-grained structured pruning, block-based pruning provides a finer pruning granularity to better preserve the DNN model accuracy. With an appropriate block size selected, the remaining computation within a block can still be parallelized on mobile device with the help of compiler. As a result, block-based pruning can achieve comparable hardware (inference) performance as coarse-grained structured pruning, under the same overall pruning rate.

Block-punched Pruning: The CONV layers prefer the tensor-based representation and computation rather than matrix-based computation used for FC layers. Inspired by block-based pruning, we develop block-punched pruning scheme tailored for CONV layers, which can be accelerated

using the same compiler optimizations. As shown in Fig. 1 (f), block-punched pruning requires pruning a group of weights at the same location of all filters and all channels within a block to leverage hardware parallelism from both memory and computation perspectives. With effective compiler-level executable code generation, high hardware parallelism (and inference acceleration on mobile) can also be achieved.

Compiler Optimizations: We develop a comprehensive, compiler-based automatic code generation framework supporting the proposed (block-punched/block-based) pruning schemes in a unified manner. It also supports other pruning schemes such as unstructured, coarse-grained, pattern-based pruning. In fact, unstructured and coarse-grained structured pruning schemes are just special cases of block-punched pruning, the former with block size 1×1 and the latter with block size of the whole weight tensor/matrix. A novel *layer fusion* technique is developed, which is critical to the efficient implementation of super-deep networks. Fast *auto-tuning* capability is incorporated for efficient end-to-end inference on different mobile CPU/GPU.

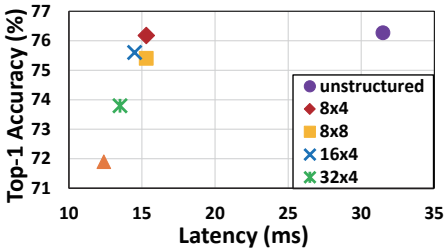

Fig. 2. Accuracy vs. Latency with different block sizes on ImageNet using ResNet-50 under uniform 6× pruning rate.

Sample Results and Block Size Determination: Fig. 2 shows example results of the accuracy vs. latency when applying block-punched pruning on ResNet-50 with different block sizes. A uniform pruning rate (i.e., 6×) and block size are adopted through all layers. Under the same pruning rate, unstructured pruning (i.e., 1×1 block size) preserves the highest accuracy but has the worst performance in latency. On the contrary, coarse-grained structured pruning (i.e., whole weight matrix as a block) achieves the lowest latency but with a severe accuracy degradation. The results of block-punched pruning show high accuracy and high inference speed (low latency) simultaneously.

The reason is that the maximum hardware parallelism is limited by computation resources. Thus, even when dividing the weights into blocks, each block's remaining weights are still sufficient to fulfill on-device hardware parallelism, especially on resource-limited mobile devices. One reasonable block size determination strategy is to let the number of channels contained in each block match the length of the vector register (e.g., 4) on the target mobile CPU/GPU to ensure high parallelism. Then determine the number of filters to be contained (e.g., 8) by considering the given design targets.

IV. Motivation of Compiler-Aware Unified Optimization Framework

Our compiler optimizations provide notable acceleration of different filter types, and support for various sparsity schemes. A key **observation** is that different filter types and sparsity schemes have different acceleration performance under compiler optimizations (when computation (MACs) is the same). The following are measured on Qualcomm Snapdragon 865 Octa-core mobile CPU of a Samsung Galaxy S20 phone.

Different Filter Types (Kernel Sizes): Fig. 3 (a) shows the latency vs. computation (MACs) of a CONV layer with different kernel sizes. We fix the input feature map to 56×56 and change the number of filters. Under the same computation, 3×3 kernels achieve the best performance, where the 1×1 kernels are the second. Because 3×3 kernels can be accelerated using Winograd algorithm, and makes it the most compiler-friendly; while 1×1 kernels result in no input redundancy in GEMM computation, which also relieves the burden on compiler optimizations.

Different Pruning Schemes: Fig. 3 (b) shows the computation speedup vs. pruning rate of a 3×3 CONV layer with different pruning schemes. We choose the input feature map size of 56×56 and 256 input and output channels. We can observe that, with compiler optimizations, fine-grained pruning schemes (i.e., pattern-based and block-punched pruning) consistently outperform the unstructured pruning and achieve comparable acceleration compared to the coarse-grained structured pruning below 5× pruning. Since, under reasonable pruning rate of fine-grained structured pruning schemes, the remaining weights in each layer are still sufficient to fully utilize hardware parallelism.

Based on the above observations, it is desirable to perform a compiler-aware network pruning and architecture search, determining the pruning scheme and pruning rate for each individual layer. The objective is to *maximize DNN accuracy satisfying an inference latency constraint* when actually executing on the target mobile device, accounting for compiler optimizations.

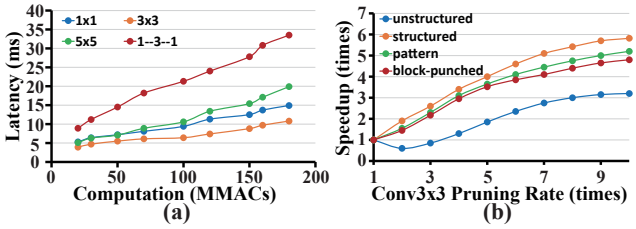

Fig. 3. (a) Latency vs. Computation with different filter types, (b) speedup vs. pruning rate with different pruning schemes.

V. Proposed Unified Network Pruning and Architecture Search (NPS) Algorithm

A. Overview of NPS Framework

Fig. 4 shows the proposed NPS framework. To take advantage of recent NAS results and accelerate the NPS process,

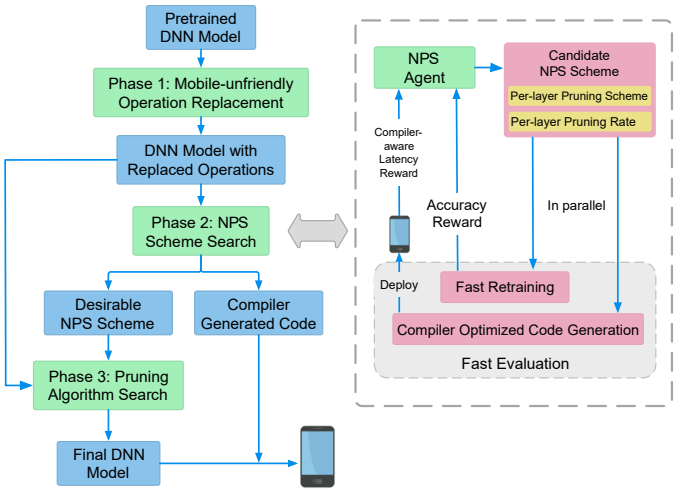

Fig. 4. Overview of the proposed NPS framework.

we start from a pre-trained DNN model, and go through three phases as shown in the figure.

Phase 1: Replacement of Mobile-Unfriendly Operations: Certain operators are inefficient to execute on mobile devices (mobile CPU and GPU). For instance, certain activation functions, such as sigmoid, swish, require exponential computation, and can become latency bottleneck on mobile inference. These unfriendly operations will be replaced by compiler-friendly alternatives such as hard-sigmoid and hard-swish, with negligible effect on accuracy.

Phase 2: NPS Scheme Search: This phase generates and evaluates candidate *NPS schemes*, defined by the collection of per-layer pruning schemes and rates, and finally chooses the best-suited one. As per-layer pruning schemes and rates are being searched, Phase 2 exhibits a much large search space, which renders representative NAS algorithms like RL-based ones ineffective. To accelerate such search, we present *a meta-modeling procedure based on RL with Bayesian Optimization* (BO). *A fast accuracy evaluation method* is developed, tailored to NPS framework.

Moreover, we incorporate the overall DNN latency constraint effectively in the reward function of NPS scheme search, ensuring that such constraint can be satisfied at the search outcome. The overall DNN latency is actually measured on the target mobile CPU/GPU based on the candidate NPS scheme currently under evaluation. We rely on actual measurement instead of per-layer latency modeling as many prior NAS work. This is because our advanced compiler optimizations incorporate a strong layer fusion beyond prior compiler work, which is critical for efficient implementation of super-deep networks, and will make per-layer latency modeling less accurate.

Phase 3: Pruning Algorithm Search: The previous phase has already determined the per-layer pruning schemes and rates, so that the compiler-generated codes can satisfy the overall latency constraint. The remaining task of this phase is to search for the most desirable pruning algorithm to perform actual pruning and train the remaining weights[1]. As the per-layer pruning rates are already determined, the candidate pruning algorithms to select from are limited to those with pre-defined per-layer pruning rates, including magnitude-based ones [13], ADMM-based algorithm [48], etc. As an extension over prior work, we generalize these algorithms to achieve different sparsity schemes with the help of group-Lasso regularization [43]. In Phase 3, we compare the resulted DNN accuracy from the candidate pruning algorithms in a few epochs, select the one with the highest accuracy, and continue a best-effort algorithm execution to derive the final DNN model and compiled codes.

B. Details of Phase 2: NPS Scheme Search

TABLE I
NPS SEARCH SPACE FOR EACH DNN LAYER

Pruning scheme	{Filter [49], Pattern-based [31], Block-punched/block-based}
Pruning rate	{ 1×, 2×, 2.5×, 3×, 5×, 7×, 10× }

1) Search Space of NPS in Phase 2: **Per-layer pruning schemes:** The NPS agent can choose from filter (channel) pruning [49], pattern-based pruning [31] and block-punched/based pruning for each layer. As different layers may have different compatible pruning schemes, we allow the NPS the flexibility to choose different pruning schemes for different layers. This is well supported by our compiler code generation.

Per-layer pruning rate: We can choose from the list $\{1\times, 2\times, 2.5\times, 3\times, 5\times, 7\times, 10\times\}$ ($1\times$ means no pruning).

2) Q-Learning Training Procedure: As per-layer pruning scheme and rate is integrated in NPS scheme search, the search space is beyond that of conventional NAS. To ensure fast search, we employ the RL algorithm Q-learning as the base technique, assisted by fast evaluation and Bayesian optimization (BO) for search speedup. The Q-learning algorithm consists of an NPS agent, states and a set of actions.

For the state of the i-th layer in a given DNN, it is defined as a tuple of pruning scheme and pruning rate i.e., $\{pruning\_scheme_i,\ pruning\_rate_i\}$, and each can be selected from the corresponding search space. We add the layer depth to the state space to constrict the action space such that the state-action graph is directed and acyclic (DAG).

For *action space*, we allow transitions for a state with layer depth i to a state with layer depth $i+1$, ensuring that there are no loops in the graph. This constraint ensures that the state-action graph is always a DAG. When layer depth reaches the maximum layer depth, the transition terminates.

Based on above-defined state $s \in S$ and action $a \in A$, we adopt Q-learning procedure [42] to update Q-values. We specify final and intermediate rewards as follows:

$$r_T = V - \alpha \cdot \max(0, h - H), \quad r_t = \frac{r_T}{T}, \tag{1}$$

[1]The above process cannot be accomplished by the fast accuracy evaluation in Phase 2 as we need to limit the number of training epochs.

where V is the validation accuracy of the model, h is the model inference speed or latency (actually measured on a mobile device), and H is the threshold for the latency requirement. Generally, r_T is high when the model satisfies the real-time requirement ($h < H$) with high evaluation accuracy. Otherwise the final reward is small, especially when the latency requirement is violated. For the intermediate reward r_t which is usually ignored by setting it to zero [3] as it cannot be explicitly measured, the reward shaping [30] is employed as shown above to speed up the convergence. Setting $r_t = 0$ could make the Q-value of s_T much larger than others in the early stage of training, leading to an early stop of searching for the agent.

We adopt the ϵ-greedy strategy [28] to choose actions. In addition, as the exploration space is large, the *experience replay* technique is adopted for faster convergence [22].

3) Fast Evaluation Methods: We develop and adopt multiple tailored acceleration strategies to facilitate fast evaluation in NPS scheme search.

One-shot Pruning and Early Stopping for Fast Accuracy Evaluation: During the accuracy evaluation process, we follow the pruning scheme and pruning rate (for a specific layer) in a candidate NPS scheme, and conduct a one-shot pruning (on the target layer) based on weight magnitude.Thisstraightforward pruning will result in accuracy degradation.But after a couple of epochs of retraining, it can distinguishthe relative accuracy of different NPS schemes

Overlapping Compiler Optimization and Accuracy Evaluation: We use compiler code generation and actual on-device latency measurement because of (i) higher accuracy than per-layer latency modeling due to layer fusion mechanism, and (ii) the fast auto-tuning capability of compiler to different mobile devices. Please note that the compiler code generation and latency measurement *do not need the absolute weight values*. Compiler code generation is much faster than DNN training (even a single epoch), and can be performed in parallel with accuracy evaluation (as accurate weight values are not needed). As a result, it will not incur extra time consumption to NPS.

4) Bayesian Predictor for Reducing Evaluations: As performing evaluation on a large amount of sampled NPS schemes is time-consuming, we build a predictor with BO [9], [39]. The NPS agent generates a pool of NPS schemes. We first use BO to select a small number of NPS schemes with potentially high rewards from the pool. Then the selected NPS schemes are evaluated to derive more accurate rewards. We reduce the evaluation of NPS schemes with possibly weak performance, thereby reducing the overall scheme evaluation effort.

We build a predictor combining Gaussian process (GP) with a Weisfeiler-Lehman subtree (WL) graph kernel [29] to handle the graph-like NPS schemes. The WL kernel compares two directed graphs in iterations. In the m-th WL iteration, it first obtains the histogram of graph features $\phi_m(s)$ and $\phi_m(s')$ for two graphs. Then it compares the two graphs with $k_{\text{base}}\big(\phi_m(s), \phi_m(s')\big)$ where k_{base} is a base kernel and we employ dot product here. The iterative procedure stops until $m = M$ and resultant WL kernel is

$$k^M_{\text{WL}}(s, s') = \sum_{m=0}^{M} w_m k_{\text{base}}\big(\phi_m(s), \phi_m(s')\big). \tag{2}$$

where w_m contains the weights for each WL iteration m, which is set to equal for all m following [38]. The *Expected Improvement* [33] is employed as the acquisition function in the work. Algorithm 1 provides a summary.

Algorithm 1 Q-learning with Bayesian Predictor Algorithm

Input: Observation data $\mathcal{D}$, BO batch size B, BO acquisition function $\alpha(\cdot)$
Output: The best NPS scheme s
for steps **do**
  Generate a pool of candidate NPS schemes $\mathcal{S}_c$;
  Select $\{\hat{s}^i\}_{i=1}^B = \arg\max_{s \in \mathcal{S}_c} \alpha(s|\mathcal{D})$;
  Evaluate the scheme and obtain reward $\{r^i\}_{i=1}^B$ of $\{\hat{s}^i\}_{i=1}^B$;
  Update Q values based on Q-learning with reward;
  $\mathcal{D} \leftarrow \mathcal{D} \cup (\{s^i\}_{i=1}^B, \{r^i\}_{i=1}^B)$;
  Update GP of BO with $\mathcal{D}$;
end for

VI. Results and Evaluation

A. Experimental Setup

In this section, we use the image classification task and ImageNet dataset to show the effectiveness of our framework.

All training processes use the SGD optimizer with a momentum rate set to 0.9 and weight decay set to 0.0005 and use the batch size of 2048 per node. The starting learning rate is set to 0.001, and the cosine learning rate scheduler is used if not specified in our paper. For Phase 1, we conduct a fast fine-tuning with 5 training epochs after replacing the mobile-unfriendly operations (only once for the entire NPS process). In Phase 2, 40 Nvidia Titan RTX GPUs are used to conduct the fast accuracy evaluation for candidate NPS schemes concurrently. Since we start from a well-trained model, we retrain 2 epochs for each candidate one-shot pruned model for fast evaluation. For each candidate model, we measure 100 runs of inference on target mobile devices and use the average value as end-to-end latency.

In Phase 3, we search the most desirable pruning algorithm including magnitude-based algorithm, ADMM-based algorithm [21], [48] and geometric median-based algorithm [16] (only for filter pruning). We adopt 100 epochs for weight pruning and 100 epochs on remaining weights fine-tuning with knowledge distillation [37].

The overall GPU days are varied based on pre-trained network and are reduced thanks to our fast evaluation and BO. For example, using EfficientNet-B0 as starting point, the overall searching time is 15 days, where Phase 1 only takes 5 epochs, and Phase 3 takes 1.5 days.

B. Evaluation Results

In Fig. 5 and 6, we compare our accuracy and latency results with representative DNN inference acceleration framework MNN, PyTorch Mobile, and TFLite. Four dense DNN

TABLE II
COMPARISON RESULTS OF NPS AND REPRESENTATIVE LIGHTWEIGHT NETWORKS.

	A. / P. Search	Params	CONV MACs	Accuracy (Top-1/5)	Latency (CPU/GPU)	Device
MobileNet-V1 [19]	N./N.	4.2M	575M	70.6 / 89.5	- / -	-
MobileNet-V2 [36]	N./N.	3.4M	300M	72.0 / 91.0	- / -	-
MobileNet-V3 [18]	Y./N.	5.4M	227M	75.2 / 92.2	- / -	-
NAS-Net-A [51]	Y./N.	5.3M	564M	74.0 / 91.3	183ms / NA	Google Pixel 1
AmoebaNet-A [34]	Y./N.	5.1M	555M	74.5 / 92.0	190ms / NA	Google Pixel 1
MnasNet-A1 [40]	Y./N.	3.9M	312M	75.2 / 92.5	78ms / NA	Google Pixel 1
ProxylessNas-R [7]	Y./N.	NA	NA	74.6 / 92.2	78ms / NA	Google Pixel 1
NPS (ours)	Y./N.	4.1M	290M	77.0 / 93.3	11.8ms / 6.7ms	Galaxy S20
NPS (ours)	Y./Y.	3.5M	201M	75.0 / 92.0	9.8ms / 5.9ms	Galaxy S20
NPS (ours)	Y./Y.	3.0M	147M	70.9 / 90.5	6.9ms / 3.9ms	Galaxy S20
NPS (ours)	Y./Y.	2.8M	98M	68.3 / 89.4	5.6ms / 3.3ms	Galaxy S20

models are used for the comparisons, which are MobileNet-V3, EfficientNet-B0, shrunk versions of EfficientNet-B0 to 70% original computation and 50% original computation. The results are tested on a Samsung Galaxy S20 smartphone using a Qualcomm Snapdragon 865 Octa-core mobile CPU and a Qualcomm Adreno 650 mobile GPU. PyTorch Mobile does not support mobile GPU, so no corresponding results. EfficientNet-B0 is used as our pretrained model.

First, without incorporating NPS, one can observe that our compiler optimizations can effectively speed up the same DNN inference, up to 46% and 141% (on MobileNet-V3), compared with the currently best framework MNN on mobile CPU and GPU, respectively. The red star shapes in the figures represent the NPS generated results under different latency constraints. Our NPS results consistently outperform the representative DNN models, and achieve the Pareto optimality in terms of accuracy and inference latency. With the highest accuracy (77.0% Top-1), the end-to-end inference time of NPS solution (290M MACs) is only 11.8ms and 6.7ms on mobile CPU and GPU, respectively. With MobileNet-V3 level accuracy (75% Top-1), our inference time (201M MACs) is 9.8ms and 5.9ms, respectively. With MobileNet-V2 level accuracy (71% Top-1), the inference time of NPS solution (147M MACs) is 6.9ms and 3.9ms, respectively. To the best of our knowledge, this is never accomplished by any existing NAS or weight pruning work.

Table II shows the model details, with representative hand-crafted and hardware-aware NAS models as references. One can observe the computation (MACs) reduction under the same accuracy compared with the prior references, thanks to the network pruning search.

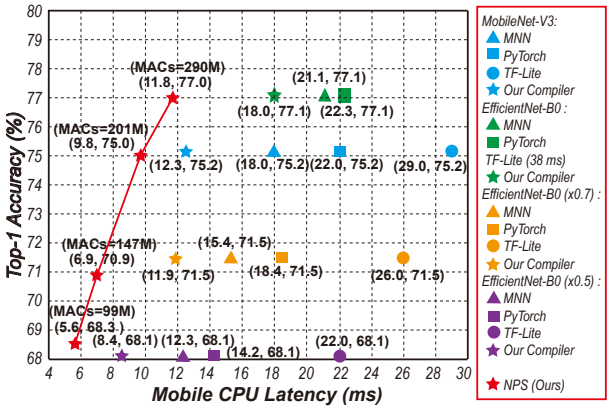

Fig. 5. Accuracy vs. Latency comparison on mobile CPU.

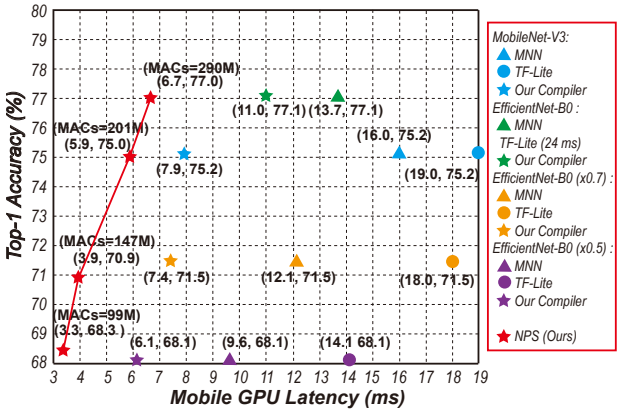

Fig. 6. Accuracy vs. Latency comparison on mobile GPU.

VII. CONCLUSION

In this work, we propose (i) a fine-grained structured pruning applicable to various DNN layers, and (ii) a compiler automatic code generation framework supporting different DNNs and different pruning schemes, which bridge the gap of model compression and NAS. We further propose NPS, a compiler-aware unified network pruning and architecture search, and several techniques are used to accelerate the searching process.

ACKNOWLEDGEMENTS

This project is partly supported by NSF under CNS-1739748 and CCF-1733701, Army Research Office (ARO) 76598CSYIP, a grant from Semiconductor Research Corporation (SRC), and Jeffress Trust Awards in Interdisciplinary Research. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF, ARO, SRC, or Thomas F. and Kate Miller Jeffress Memorial Trust.

REFERENCES

[1] https://www.tensorflow.org/mobile/tflite/. 3

[2] https://github.com/alibaba/MNN. 3

[3] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2017. 6

[4] James Bergstra, Daniel Yamins, and David Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *International conference on machine learning*, pages 115–123, 2013. 2

[5] Andrew Brock, Theodore Lim, James M Ritchie, and Nick Weston. Smash: one-shot model architecture search through hypernetworks. *arXiv preprint arXiv:1708.05344*, 2017. 2

[6] Han Cai, Tianyao Chen, Weinan Zhang, Yong Yu, and Jun Wang. Efficient architecture search by network transformation. *arXiv preprint arXiv:1707.04873*, 2017. 2

[7] Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332*, 2018. 1, 2, 3, 7

[8] Tianqi Chen et al. Tvm: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018. 3

[9] Yutian Chen, Aja Huang, Ziyu Wang, Ioannis Antonoglou, Julian Schrittwieser, David Silver, and Nando de Freitas. Bayesian optimization in alphago. *arXiv preprint arXiv:1812.06855*, 2018. 6

[10] Peiyan Dong, Siyue Wang, Wei Niu, Chengming Zhang, Sheng Lin, Zhengang Li, Yifan Gong, Bin Ren, Xue Lin, Yanzhi Wang, et al. Rtmobile: Beyond real-time mobile acceleration of rnns for speech recognition. *arXiv preprint arXiv:2002.11474*, 2020. 1

[11] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic network surgery for efficient dnns. In *Advances in neural information processing systems (NeurIPS)*, pages 1379–1387, 2016. 2

[12] Zichao Guo, Xiangyu Zhang, Haoyuan Mu, Wen Heng, Zechun Liu, Yichen Wei, and Jian Sun. Single path one-shot neural architecture search with uniform sampling. In *European Conference on Computer Vision*, pages 544–560. Springer, 2020. 2

[13] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *International Conference on Learning Representations (ICLR)*, 2016. 2, 5

[14] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems (NeurIPS)*, pages 1135–1143, 2015. 1, 2

[15] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 784–800, 2018. 1

[16] Yang He, Ping Liu, Ziwei Wang, Zhilan Hu, and Yi Yang. Filter pruning via geometric median for deep convolutional neural networks acceleration. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4340–4349, 2019. 2, 6

[17] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 1389–1397, 2017. 2

[18] Andrew Howard et al. Searching for mobilenetv3. In *ICCV*, 2019. 1, 7

[19] Andrew Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv:1704.04861*, 2017. 7

[20] Nicholas D Lane et al. Deepear: robust smartphone audio sensing in unconstrained acoustic environments using deep learning. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 283–294. ACM, 2015. 3

[21] Tuanhui Li et al. Compressing convolutional neural networks via factorized convolutional filters. In *CVPR*, 2019. 2, 6

[22] Long-Ji Lin. *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, USA, 1992. 6

[23] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018. 2

[24] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the value of network pruning. *arXiv preprint arXiv:1810.05270*, 2018. 1, 2

[25] Xiaolong Ma, Fu-Ming Guo, Wei Niu, Xue Lin, Jian Tang, Kaisheng Ma, Bin Ren, and Yanzhi Wang. Pconv: The missing but desirable sparsity in dnn weight pruning for real-time execution on mobile devices. In *Thirty-Four AAAI Conference on Artificial Intelligence*, 2020. 1, 2, 3

[26] Xiaolong Ma, Wei Niu, Tianyun Zhang, Sijia Liu, Fu-Ming Guo, Sheng Lin, Hongjia Li, Xiang Chen, Jian Tang, Kaisheng Ma, et al. An image enhancing pattern-based sparsity for real-time inference on mobile devices. *arXiv preprint arXiv:2001.07710*, 2020. 2, 3

[27] Chuhan Min, Aosen Wang, Yiran Chen, Wenyao Xu, and Xin Chen. 2pfpce: Two-phase filter pruning based on conditional entropy. *arXiv preprint arXiv:1809.02220*, 2018. 1

[28] Volodymyr Mnih et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015. 6

[29] Christopher Morris, Kristian Kersting, and Petra Mutzel. Glocalized weisfeiler-lehman graph kernels: Global-local feature maps of graphs. In *2017 IEEE International Conference on Data Mining (ICDM)*, pages 327–336. IEEE, 2017. 6

[30] Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, volume 99, pages 278–287, 1999. 6

[31] Wei Niu, Xiaolong Ma, Sheng Lin, Shihao Wang, Xuehai Qian, Xue Lin, Yanzhi Wang, and Bin Ren. Patdnn: Achieving real-time dnn execution on mobile devices with pattern-based weight pruning. *arXiv preprint arXiv:2001.00138*, 2020. 1, 2, 3, 5

[32] Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018. 2

[33] Chao Qin, Diego Klabjan, and Daniel Russo. Improving the expected improvement algorithm. In *Advances in Neural Information Processing Systems*, pages 5381–5391, 2017. 6

[34] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, volume 33, pages 4780–4789, 2019. 2, 7

[35] Binxin Ru, Xingchen Wan, Xiaowen Dong, and Michael Osborne. Neural architecture search using bayesian optimisation with weisfeiler-lehman kernel. *arXiv preprint arXiv:2006.07556*, 2020. 2

[36] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018. 7

[37] Zhiqiang Shen and Marios Savvides. Meal v2: Boosting vanilla resnet-50 to 80%+ top-1 accuracy on imagenet without tricks. *arXiv preprint arXiv:2009.08453*, 2020. 6

[38] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(77):2539–2561, 2011. 6

[39] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012. 6

[40] Mingxing Tan et al. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2820–2828, 2019. 1, 3, 7

[41] Mingxing Tan and Quoc V Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*, 2019. 1

[42] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989. 5

[43] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In *Advances in neural information processing systems (NeurIPS)*, pages 2074–2082, 2016. 1, 2, 5

[44] Bichen Wu et al. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *CVPR*, 2019. 1, 3

[45] Lingxi Xie and Alan Yuille. Genetic cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1379–1388, 2017. 2

[46] Mengwei Xu et al. Deepcache: Principled cache for mobile deep vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 129–144. ACM, 2018. 3

[47] Ruichi Yu et al. Nisp: Pruning networks using neuron importance score propagation. In *CVPR*, 2018. 2

[48] Tianyun Zhang, Shaokai Ye, Yipeng Zhang, Yanzhi Wang, and Makan Fardad. Systematic weight pruning of dnns using alternating direction method of multipliers. *arXiv preprint arXiv:1802.05747*, 2018. 2, 5, 6

[49] Zhuangwei Zhuang et al. Discrimination-aware channel pruning for deep neural networks. In *NeurIPS*, 2018. 1, 5

[50] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2017. 1, 2

[51] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, pages 8697–8710, 2018. 7